A FRAMEWORK FOR AUTOMATED MANAGEMENT OF EXPLOIT TESTING ENVIRONMENTS

A Thesis
Presented to
The Academic Faculty

by

Kevin Daniel Flansburg

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Computational Science and Engineering

Georgia Institute of Technology
May 2016

A FRAMEWORK FOR AUTOMATED MANAGEMENT OF EXPLOIT TESTING ENVIRONMENTS

Approved by:

Professor Taesoo Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Manos Antonakakis
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Duen Horng (Polo) Chau
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: November 23, 2015

To my mother and father, Barrett and Lori Flansburg,

for their never-ending support and love.

ACKNOWLEDGEMENTS

I would like to thank Taesoo Kim for his support, wisdom, and prevailing confidence in me. I feel very fortunate to have been able to work with him. I would like to thank my committee members, Polo Chau and Manos Antonakakis for their guidance. I would like to thank Yeongjin Jang and Insu Yun for their assistance on this project, their expertise was immensely helpful. You are all truly inspiring. Finally, I would like to thank my close friends Lincoln, Billy, Sterett, and Ron for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis outlines the design and implementation of a framework for automating the process of building virtual test environments for security research. It discusses the metadata needed to fully reproduce identical environments on separate host systems. A software library capable of building such environments from this metadata, and performing tests within these environments, is implemented. Finally, the capabilities of this tool are evaluated by using it in a variety of test configurations.

CHAPTER I

INTRODUCTION

To demonstrate working exploits or vulnerabilities, people often share their findings as a form of proof-of-concept (PoC) prototype on online forums or community websites. For exploit authors, such demonstrations are a great vehicle to bring their fame, and in general, these are useful, lively resources for others to learn about real vulnerabilities and state-of-the-art exploitation techniques. Unfortunately, the shared PoC exploits are seldom reproducible, in part because they are often not thoroughly tested, but largely because authors lack a formal way to specify the tested environment and its dependencies. For example, Figure 1 shows a typical practice of describing dependencies and tested environments when sharing a PoC exploit on community websites (e.g., `https://exploit-db.com`). To demonstrate that the shared exploit is working, the author often shares three logical aspects of the exploit along with the exploit itself: tested environments and dependencies; execution method and possible configuration; and validation to check a successful exploitation.

This example code snippet shows a PoC exploit attacking the GHOST vulnerability [1]. The GHOST vulnerability, officially known as CVE-2015-0235, is a buffer overflow bug in the `gethostbyname()` function of `glibc`, which was introduced in November, 2000 [40]. Although the GHOST bug had existed for over three years, it was hard to replicate when the bug was announced. Not only does it require a specific version of `glibc` (from `glibc-2.2` up to `glibc-2.17`), but, at the time that it was announced, all distributions had already dynamically updated or fixed the bug in their main package repositories, requiring testers to roll back their version of `glibc`. Therefore, to reproduce the public GHOST exploit, one needs to replace the system-wide `glibc` library, which may end up requiring the re-installation of numerous packages in order to resolve the cascading dependencies.

```
# Exploit Title:  [Exim ESMTP GHOST DoS PoC Exploit]
# Date:           [1/29/2015]
# Exploit Author: [1N3]
# Vendor Homepage:[www.exim.org]
# Version:        [4.80 or less]
# Tested on:      [debian-7-7-64b]
# CVE:            [2015-0235]

#!/usr/bin/python
# Exim ESMTP DoS Exploit by 1N3 v20150128
# CVE-2015-0235 GHOST glibc gethostbyname
# buffer overflow. http://crowdshield.com
#
# USAGE: python ghost-smtp-dos.py <ip> <port>
#
# Escape character is '^]'.
# 220 debian-7-7-64b ESMTP Exim 4.80 ...
# HELO
# 000000000000000000 ...
# Connection closed by foreign host.
#
# user () debian-7-7-64b:~# dmesg
# ...
# exim4[2562]: segfault at 7fabf1f0ecb8
#   ip 00007fabef31bd04
#   sp 00007fffb427d5b0 error 6 in
#   libc-2.13.so[7fabef2a2000+182000]
...
```

① ***Dependencies***

② ***Execution***

③ ***Validation***

Figure 1: An snippet of an exploit PoC for the GHOST vulnerability that targets the Exim mail server (`https://www.exploit-db.com/exploits/35951/`). Exploit writers, typically, leave comments on ① tested environment and dependencies, ② how to execute, and ③ how to check if it succeeds, which are shaded in gray on the figure.

Table 1 shows possible problems encountered when reproducing PoC exploits. Such practical barriers make exploits difficult to develop, faithfully test, or correctly share with the community. Non-reproducible exploits have limited potential of being utilized in various other applications that could benefit from publicly available PoC exploits. For example, the reproducible PoC exploits can be used to evaluate existing mitigation techniques to see how effective each technique is in defeating real exploits.

To solve this problem, we present XSHOP, a framework and infrastructure to describe and share environments and dependencies for running PoC exploits in a formal way. XSHOP's flexible design enables new possibilities for utilizing these real, reproducible exploits. As a demonstration, we build *five* practical use cases on top of XSHOP: easy community sharing and reproduction of exploit PoC, as a security benchmark suite, for pen-testing, regression testing, and to aid in authoring PoC. We design and implement these applications by extending the XSHOP framework and demonstrate their effectiveness with *twelve* real world

Table 1: Problems when running PoC exploits of wide-spread vulnerabilities.

CVE (Nickname)	Difficulties Encountered
CVE-2015-0235 (GHOST)	Wide adoption of partially fixed `glibc` (2.17 to 2.18) among various distributions. Required downgrade of major library.
CVE-2014-{6271,6277, 6278,7169,7186} (Shellshock)	Fixed across five different patches. (see Figure 3)
CVE-2013-6629 (libjpeg memory leak)	Highly non-standard installation. No actual PoC script released.
CVE-2014-0160 (Heartbleed)	Configuring two machine environment: an exploit runner and an SSL server.
CVE-2010-{4258,3849, 3850} (Full Nelson)	Configuring machine with specific, old kernel. Ubuntu kernel patched before upstream. Handling deadlock after success.

PoC exploits against well-known bugs that include GHOST, Shellshock and Heartbleed in userspace and Full Nelson in the Linux kernel. We believe that XSHOP is not limited to academic research, but has a potential to be grown as a community-wide knowledge base because the proposed practices bring an immediate incentive to exploit writers (see §9).

In this paper, we make the following three contributions:

1. We design and implement XSHOP as an open-source framework and infrastructure to develop, test and share exploits in a reproducible manner.

2. We demonstrate XSHOP's effectiveness with five practical use cases: portable and reproducible PoC exploits, security benchmark suite, pen-testing utility, regression testing tool, and exploit authoring environment.

3. We evaluate XSHOP and its applications with a representative set of exploits that attack 5 vulnerabilities in applications and 7 vulnerabilities in the Linux kernel. We believe XSHOP has significant potential to become a common practice to specify and share PoC exploits, and also to grow as a community-driven knowledge base.

The rest of the paper is organized as follows. In §2, we discuss our motivation for developing XSHOP. §3 presents its high-level overview. §4 shows a number of different use

cases we built on top of XSHOP. §5 describes in detail our goals and XSHOP's design. §6 shows XSHOP's implementation. §7 illustrates a workflow by using the Heartbleed bug as a running example. §8 evaluates XSHOP and the proposed applications by using real PoC exploits. §9 discusses XSHOP's limitation, envisions its potential as a community-driven knowledge base, and provides XSHOP's incentives for exploit authors. §10 compares our approach with previous work, and finally, §11 concludes.

CHAPTER II

MOTIVATION

In this section, we outline three motivating scenarios for reproducible exploit testing and sharing. For each scenario, we describe existing difficulties and ways in which XSHOP can mitigate these issues.

2.1 Reproducing Exploit Proof of Concept

When users set out to reproduce a published exploit PoC, they can be met with a number of challenges. They must re-create the test environment, often from incomplete information. This configuration can be further complicated by requiring specific software versions, including system libraries or kernels.

Occasionally, the actual exploit is not published, only the attack vector. The user may not be familiar enough with the vulnerability to follow this easily. If a script is provided, it may have additional dependencies, or rely on a specific memory layout to work. Finally, verification of the exploit may not be straightforward, the configuration process of the author's test environment may have inadvertently introduced the vulnerability, but this cannot be seen to others.

This process of reproducing the original exploit PoC can be very time-consuming, and must be repeated by any user who wishes to work with the PoC. Similarly, the original author can benefit from automatic provisioning of test environments to their specification.

2.2 Research Applications

Currently, because of the time involved in reproducing many exploits, researchers find it too cumbersome to validate their work with real world examples. Many papers make use of artificially generated exploit examples such as RIPE [51]. These artificial examples are

certainly useful to evaluate textbook cases, but real world examples lend greater credibility to the results because they often make use of corner cases or more creative attack vectors. Ideally, published PoCs should be easily used in strengthening other areas of security research.

To expand on this, an important component of experimentation is the control of test variables. In a conventional scenario, one may be forced to manually configure several test environments, attempting to reduce the variation between them. The ability to perfectly re-create test environments, with some ability to introduce variation in a controlled manner, would greatly improve the rigor of large scale experiments.

2.3 Isolation from Host System

Testing frequently involves running exploit code from untrusted sources online, installing known-to-be-vulnerable software, and intentionally downgrading to make the system itself vulnerable. It is certainly recommended as best practice that these tests are performed in an isolated environment; however, the overhead and repetitiveness of configuring such environments discourages this. A tool for automatically building many consistent test environments would greatly reduce this overhead, and encourage users to tinker with potentially dangerous exploits in a safe manner without much hassle.

CHAPTER III

XSHOP OVERVIEW

Figure 2 shows an overview of XSHOP. XSHOP consists of three components which are the Build Manager, Exploit Driver and Hook Manager. The Build Manager supplies an isolated and uniform environment to reproduce a given PoC. It manages virtual environments, installs all dependent packages and a vulnerable target program, and launches the program if needed. It is similar to the exploit writer's comments about dependencies, however it automatically constructs the environment for the user. In the case of the Heartbleed PoC the Build Manager creates two virtual systems because Heartbleed is an attack with a server-client model. In one system, the Build Manager compiles and configures the desired version of OpenSSL. The other system connects to this server as a client, and attempts to exploit it.

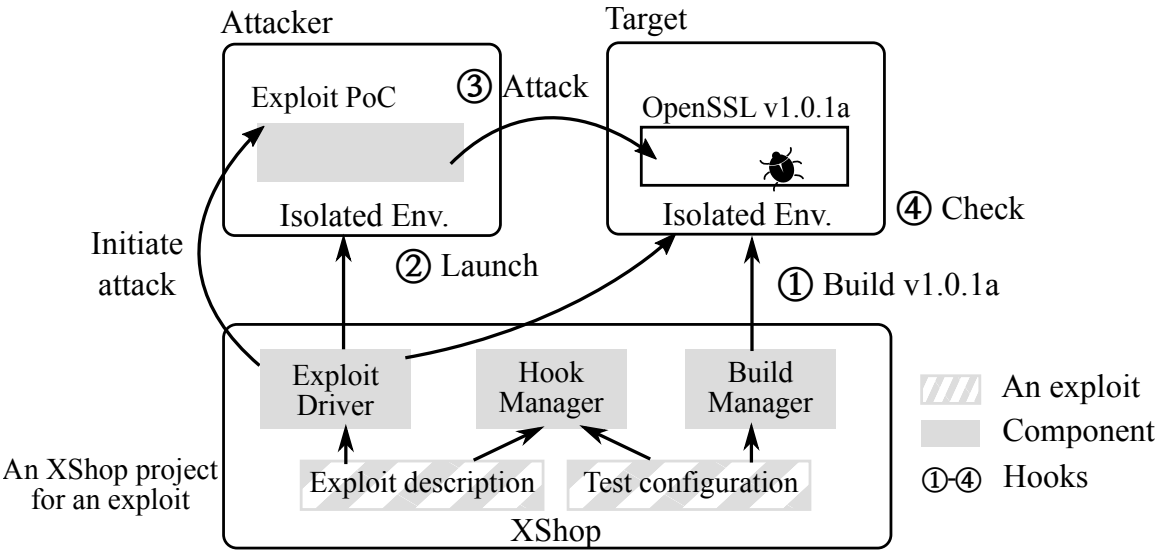

Figure 2: The overview of XSHOP's design. To run an exploit for Heartbleed, XSHOP first prepares a vulnerable virtual environment image (Target in the figure), after building a specific version of OpenSSL (v1.0.1a). Then, it launches two virtual systems, namely Attacker and Target, and runs the actual PoC to exploit the Heartbleed bug in the OpenSSL library. In every phase of XSHOP, users can place hooks (①-④) to configure environment settings or examine side-effects of the attack.

The Exploit Driver executes the exploit and validates whether the exploit was successful. A success of the exploit can be verified in multiple ways. If an exploit is a memory leak vulnerability, the number of returned bytes can be counted. If it is a control flow attack, we can check whether a shell is spawned. In case of a privilege escalation attack, the `uid` of the resulting process can be checked. The exploit driver corresponds with the execution and validation chapter in Figure 1. In the case of Heartbleed (a memory leak vulnerability), the exploit driver runs an exploit script in the attacking system, which sends crafted packets to the target. It then receives output from the target and validates whether the exploit was successful by measuring the returned packet size.

Lastly, the Hook Manager is responsible for placing hooks in multiple positions during the test. By using these flexible hooks, XSHOP can use an exploit not only for replaying the test, but also in many other applications. In §4, we will discuss several applications of XSHOP that these flexible hooks provide.

CHAPTER IV

USE CASES

XSHOP's flexibility allows it to be used in many different ways. It gives users the ability to manage variables and automate the provisioning of test environments. In this chapter we detail just a few of the many ways in which XSHOP can assist in security research.

4.1 Portable Exploit Evaluation Environment

As mentioned in §2, exploit writers are required to manually configure test environments by carefully configuring physical or virtual machines with vulnerable software, a time consuming, risky, and not easily reproducible process. With XSHOP, much of the repetitive steps involved in this task can be automated, and authors only need to focus on the particulars of their specific test. The authors can then publish these environments for others to use with minimal additional effort.

Currently, when an exploit is published, it is accompanied by a vague description of the test scenario and conditions. Each person who wishes to verify the results or review the integrity of the test must reproduce this environment manually, perhaps making a lot of inferences. With XSHOP, users simply download the few metadata files which describe the test environment and XSHOP automatically builds an identical environment which does not depend upon a specific host configuration. XSHOP therefore allows sharing of portable environments such that many users across the web can quickly be on the same page whilst evaluating a newly discovered vulnerability. This facilitates a more involved and stronger community.

4.2 Benchmark Suite to Evaluate Mitigation Schemes

A number of compiler-based mitigation techniques such as AddressSanitizer [44], CPI [25], etc. [50, 45, 47, 27, 28] have been developed to prevent various types of vulnerabilities. Authors of these techniques seek to validate their effectiveness via automated testing. Many authors use automatic test generators such as RIPE [51], but often find it impractical to test using real world exploits. This is due to the overhead involved in configuring test environments, as each exploit may have a different set of conditions under which it succeeds. Testing with real world examples can be very useful because they cover a wider and more imaginative range of ways in which a vulnerability may be exploited than those used by automatically generated tests.

XSHOP can be a powerful tool in these circumstances. Exploits published via XSHOP come with their own test environment, eliminating this overhead and allowing authors to focus on more thorough testing of their mitigation schemes. XSHOP additionally offers very flexible control of the test procedure. This can allow for applying binary instrumentation (e.g. Oxymoron [3], BinCFI [54]) or dynamic instrumentation (e.g. PIN [31], DynamoRIO [7]) techniques after compilation, but before the exploit is attempted, and actions to determine the results within the test environment after the exploit attempt.

As a result, a developer of one of these techniques can assemble a battery of sample vulnerabilities and quickly gather data on their tool's effectiveness in guarding against such attacks. We present such an evaluation of the mitigation strategies listed above in §8.

4.3 Penetration Testing

Penetration testing (pen-testing) involves examining real systems for vulnerabilities. Currently, it is very difficult to exactly duplicate target production machines, so pen-testers are forced to take on the risk of attacking the actual machines, or attempt to recreate the machine's configuration manually.

XSHOP offers the user the ability to test for vulnerabilities in isolation. It can package an

existing system's environment into a container, allowing the user to conduct the test on a duplicate of the original system. It also allows for testing of production containers or virtual machine images. If desired, XSHOP can also launch the attack against a real remote host, as with traditional pen-testing.

For system administrators, when faced with a zero-day exploit, they must quickly determine which versions of the software used across their fleet are vulnerable in order to minimize potential damage. Previously, they had to rely on the security community to exhaustively test the software's configurations or provision testing environments to conduct the analysis themselves. In such time critical situations, this delay presents a serious risk.

XSHOP allows users to immediately test for zero-day vulnerabilities that are published as an XSHOP project, with no additional effort by the original author. This allows pen-testers and system administrators to quickly assess their vulnerability to newly published exploits, without spending critical time setting up a test environment or waiting for a test to be implemented in their testing suite of choice.

4.4 Analyzing Vulnerable Versions

When new exploits are discovered, it is important to evaluate the impact of the vulnerability in terms of number of at-risk library versions. Application developers can explore the vulnerable chapter of code using tools such as `git bisect`. With a proper understanding of the vulnerability, they can quickly identify when a bug was introduced, and identify vulnerable releases.

XSHOP can facilitate a similar exploration of the software, making it possible for the security community to identify which versions of the software present in the population are vulnerable. This is true even when there is only access to the installation package or if there is no knowledge of how the exploit works internally. XSHOP can automatically determine which versions of the affected software must be patched, without waiting for this information to be reported by the developer.

In §8, Figure 3, we show the results of such an exhaustive search in analyzing Bash from version 3.0 to 4.2 against the Shellshock vulnerability. The figure shows that the vulnerability was there before Bash 3.0 was developed, and sequentially patched for the disclosed Common Vulnerabilities and Exposures (CVEs). This ability to quickly figure out the attack surface and immediately take measures to mitigate the impact of the vulnerability is critical for system security when faced with zero-day exploit situations.

4.5 Regression Testing

When an upstream developer is ready to push the release version of the software, they frequently run a full battery of unit and integration tests before submitting their changes. Some projects which are especially concerned with security have additional test suites to test against previously disclosed exploits of their software (e.g. Google Chrome [16], Firefox [32], etc.), however, this is uncommon. With XSHOP, it is trivial for developers to create a battery of test cases from published vulnerabilities and run them against a new release in an automated manner to ensure that bugs have not been inadvertently reopened.

CHAPTER V

DESIGN

Motivated by the complexities of writing, sharing, and reproducing exploit PoC, we present XSHOP, a tool and infrastructure that help exploit writers to set up a reproducible development environment for PoC. In this section, we describe our goals and XSHOP's design in detail.

5.1 Design Goals

To generate reproducible, yet flexible testing environments for PoC, XSHOP has four important design goals as follows:

1. **Portability.** In order to reduce the time the community spends reproducing shared test results, we want to implement XSHOP to build testing environments from minimal, easily shared metadata files. This prevents the uploading and downloading of large images. Additionally, these files should be logically organized so that it is easy to see how the test environment is configured for a given PoC.

2. **Automatic Provisioning.** Using this metadata, XSHOP should be able to fully provision the test environment and run the exploit test, without assistance from the user. This eliminates repetitive housekeeping tasks and allows the user to focus on their research.

3. **Isolated Environments.** The test environments for PoC exploits should be isolated from the host. This reduces the risk to the user from testing vulnerable software with test code acquired from possibly untrusted sources.

4. **Flexibility.** Test environments should be identical between runs, but should allow controlled variations. This permits rigorous testing of the impacts of such variations

on exploit success, minimizing the impacts of other environment variables. Most importantly, XSHOP should support various kinds of PoC exploits that include remote, man-in-the-middle, and kernel attacks.

5.2 Project Description Format

To satisfy our requirement of portability, our design must include a compact project format for sharing all the details of a PoC with the community. This format must include:

1. **Dependencies.** The exploit author must have a way to specify build and runtime dependencies for the test, so that they are automatically resolved for the user.

2. **Environment.** The project must include all information necessary for XSHOP to reproduce the test environment consistently across different hosts.

3. **PoC.** The exploit author must be able to store custom scripts which carry out the exploit PoC.

4. **Validation.** The project must allow the exploit author to flexibly define a test procedure for confirming the success of the PoC.

Once a project description is specified by an author, the rest of the tasks are coordinated by XSHOP: setting up the testing environment, carrying out the PoC, and validating its result. Furthermore, a variety of applications (see §4) can be built by using XSHOP due to its flexible architecture, which we will discuss in the next section.

5.3 XSHOP Components

XSHOP consists of three major components, as shown in Figure 2: exploit driver, hook manager, and build manager. These components allow XSHOP to construct an isolated environment for reproducibly testing PoC exploit.

5.3.1 Build Manager

The build manager interacts with the isolation provider (e.g. Linux containers, virtual machines, etc.) depending on requirements specified by the author. It first populates one or more images per each role (e.g., attacker or victim) for the isolation provider, and initiates the building process to set up the test environment with proper configurations.

5.3.2 Exploit Driver

The exploit driver orchestrates the test procedure inside of the running test environment. Once the dependencies are resolved, it first runs the PoC exploit and collects its results, such as standard output and return code, for further investigation. In XSHOP, the return code to indicate the success of a PoC exploit is well-structured (i.e., `XSHOP_SUCCESS` or `XSHOP_FAILED`).

5.3.3 Hook Manager

The hook manager coordinates any custom activities which users *other than the exploit authors* may wish to perform while testing a PoC exploit with XSHOP. For example, users can easily replace the compilation options during the build process to evaluate compilation-based defense techniques with the PoC. §8 and §4 show XSHOP's applications in detail.

5.4 Controlled Variations

To allow controlled variations during the build or test procedures, XSHOP accepts a custom configuration, called . It is similar in spirit to a shell script, but allows variations in a controlled form to make sure all commands run as expected. In structure, XShopFile consists of *verbs* that describe predefined actions (e.g., `RUN` and `ADD`), *variables* that represent a set of permutations for build configurations (e.g., `library` and `version`), and minimal control logic (e.g., a for-loop). It supports five verbs in total, as summarized in Table 2, and is expressive enough to support all PoC exploits used for evaluations in §8.

Table 2: A list of supported verbs in XShopFile.

VERB [argument+]	**Function**
`RUN` [`command`]	Runs `command` in the virtual environment.
`ADD` [`path1`] [`path2`]	Copies the file at `path1` within the build context to `path2` in the virtual environment.
`WORKDIR` [`path`]	Performs all subsequent commands within from the directory specified by `path`.
`KERNEL` [`version`]	Restarts the virtual environment using the specified kernel `version`, if supported.
`FROM` [`image`]	Specifies the `image` from which the virtual environment is built. Multiple providers can be supported by using FROM_[PROVIDER] to specify the image for each.

CHAPTER VI

IMPLEMENTATION

XSHOP provides a number of Python modules and classes for easy extension, and consists of approximately 1,500 lines of code. The complexity of each major component can be found in Table 3.

Table 3: Components of XSHOP and their complexities in terms of their lines of code.

Component	Lines of code
Command Line	42 lines of Python
XSHOP Internals	941 lines of Python
Default Project Files	108 lines of XSHOPFile, test environment configuration, and validation script boilerplate.
Docker Interface	184 lines of Python
Vagrant Interface	255 lines of Python
Total XSHOP	1530 lines of Python

6.1 Project Structure

A basic XSHOP project is shown in Example 1. These files are broken up into several responsibilities: project settings, test environment, and test procedure. This set of files allows the user to fully and flexibly define the test environment and procedure, and is typically only a few kilobytes. The various formats are chosen to maximize the clarity of the test environment and procedure for users, and isolate the responsibilities of each file as much as possible.

6.2 Isolation of Testing Environment

XSHOP is designed to allow easy implementation of any isolation provider. It defines a small set of methods (e.g. `build_environment()`, `run_function()`, `launch_test()`, etc.) so that any provider can interface with XSHOP using a small Python class. Such classes have been

```
|-config.yml            ; Project configuration
|-containers            ; Test system build contexts
| |-attacker
| | |-XShopFile         ; Attacker build instructions
| |-target
| | |-XShopFile         ; Target build instructions
|-test-environment.yml  ; Test environment configuration
|-packages              ; Stores installation packages
|-source                ; Stores source archives
|-test                  ; Stores test dependencies
| |-xshop_test.py       ; Test procedure
```

Example 1: A typical XSHOP project tree.

implemented to allow XSHOP to support containers [8] via Docker [13] and VirtualBox [37] based virtual machines via Vagrant [18].

6.3 Source Management

XSHOP automatically manages installation files. Users are encouraged to place these files in the `source` and `packages` directories, and use specific naming conventions. When a test is run, the correct installation files are automatically made available in the build context.

6.4 XShopFile

To provide the aforementioned functionalities of XShopFile in §5, we extend Docker's configuration, called Dockerfile [14], by implementing our own parser. When reading a file, a dictionary of template values is passed to the parser, which then utilizes Jinja2 [43] to apply the template to the file contents. This also allows for control flow and loops within the file. Example 2 illustrates these features in an example XSHOPfile.

6.5 API

To allow for more complicated experimentation, XSHOP exposes an API consisting of the `TestCase` class, which represents a single test with fixed variable values, and a `Trial` class, which allows managing full experiments in which multiple parameters can be varied. The use of this API is demonstrated in §7.

```
FROM_DOCKER xshop:base_test_image
FROM_VAGRANT ubuntu/precise64

{% for d in builddependencies %}
RUN apt-get -y install {{ d }}
{% endfor %}

ADD {{ library }}-{{ version }}.tar.gz /home/
WORKDIR ~/ {{ library }}-{{ version }}
RUN ./configure --prefix=/usr
RUN make
RUN make install

WORKDIR ~/
```

Example 2: An example of XShopFile to specify the build process of a XSHOP project; the code snippet shows a compilation procedure for OpenSSL.

6.6 Pen-Testing

To increase the utility of PoC exploits, XSHOP allows tests to be run on both remote hosts and container or virtual machine images. Such testing is made possible with only minor manipulations to the test environment during the build phase, and easily invoked using the API.

CHAPTER VII

A CASE STUDY ON THE HEARTBLEED EXPLOIT

To further motivate the use of XSHOP, this section describes in detail the difficulties involved for both exploit authors and users wishing to work with existing PoC, and how the workflow can be improved with XSHOP. We describe this process for the Heartbleed [39] vulnerability.

CVE-2014-0160, better known as Heartbleed, is a vulnerability in the OpenSSL library which is triggered by sending a crafted SSL heartbeat packet. The library responds with a packet which contains too much data, including data from leaked regions of memory. A number of researchers have demonstrated that it is possible to obtain a server's private SSL key through this vulnerability [35], indicating that this vulnerability is particularly critical. This discovery forced a massive effort to patch OpenSSL and replace SSL keys across the Internet, and indicated that it had been possible to decrypt SSL traffic with little trace since the introduction of OpenSSL 1.0.1.

7.1 Exploit Authors

An author, wishing to write an exploit PoC for this suspected vulnerability, would desire to set up a test environment with two systems, one running an OpenSSL server and the other attacking, to demonstrate the remote exploit. This involves the uninteresting task of booting two virtual machines and configuring one to respond to SSL requests using the desired version of OpenSSL. To expand testing to additional versions of OpenSSL, or other configurations, this task must be repeated in its entirety.

XSHOP automatically configures the virtual environment for the author, and allows them to start with a fresh environment for each test. The author need only worry about the idiosyncrasies of configuring OpenSSL. The author can edit just a few lines of straightforward configuration files and be up and running in seconds. Furthermore, switching to a different

library version for testing is trivial.

7.2 Community Members

A user wishing to evaluate a Heartbleed PoC published as an XSHOP project simply downloads the few configuration files and is up and running with just a few keystrokes, completely eliminating the time spent setting up their version of the test environment to match the author's. Furthermore, by inspecting the configuration file, they can see exactly how the test is constructed, to verify that the original author did not inadvertently introduce the vulnerability.

7.3 The XSHOP *Workflow*

To demonstrate the minimal additional effort required by the exploit author, this section outlines a typical process for authoring a PoC with XSHOP. We begin by creating a new project with a command:

```
$ xshop new openssl Heartbleed
```

This creates a default XSHOP project from which to start. The next step is to edit the additional metadata for the project in `config.yml`. This file includes information such as where XSHOP can acquire source files used for the test and the default parameters used to demonstrate the PoC.

Example 3 shows the `config.yml` file for the Heartbleed test. The simple proof of concept test for the project checks the version before the patch (`1.0.1f`) and the version right after the patch (`1.0.1g`) for vulnerability. The file lists two URL's for source files. Next, we list the public keys used to sign the files that the URLs point to. This data allows XSHOP to automatically fetch the required source files so that new users may run the test. This is achieved with `xshop pull`.

Next, the we tweak the XShopFile used to build the target OpenSSL server to match the procedure for compiling and installing OpenSSL. The default `docker-compose.yml`

```
constants:
  build-dependencies: []
  dependencies: []
  library: openssl
  install_type: source
  provider: docker
variables:
  version:
    - 1.0.1f
    - 1.0.1g
source:
  - http://openssl.org/source/old/1.0.1\
          /openssl-{1.0.1f,1.0.1g}.tar.gz
public_keys:
  - F295C759
```

Example 3: Project configuration file (`config.yml`) for Heartbleed.

describes a client/server attack scenario, and so this file does not need to be modified.

We place basic self-signed certificates in the `test` folder so that they are available to the OpenSSL server inside of the environment. Finally, we modify the test script to first launch the server in the target machine, then launch the attack script supplied by the author from the attacking machine. This script is shown in Example 4.

```
import subprocess
import heartbleed

def run(T):
    T.run('target','start_server')
    T.wait_socket("target","localhost", 443)
    T.run('attacker','run_exploit')

def start_server():
    subprocess.Popen(["openssl" " s_server"
        " -key" " server.key"
        " -cert" " server.cert5"
        " -accept" " 443", " -www"],
        stdout=subprocess.PIPE, stdin=subprocess.PIPE,
        stderr=subprocess.PIPE, shell=True)

def run_exploit():
    if heartbleed.main():
        return XSHOP_SUCCESS
    else:
        return XSHOP_FAILED
```

Example 4: An example test script that runs an OpenSSL server and coordinates the Heartbleed exploit.

Within the `run()` function, we are given an object which allows us to script which functions should be called, and in what virtual system. Only functions which return `XSHOP_SUCCESS` or `XSHOP_FAILED` impact the results of the test. Helper functions, such as `start_server()` which return normally are ignored unless there is an error. Here, the

result of a separate exploit script is used to determine the success of the attempt.

Having written a test procedure and validation criterion, the author is now free to focus on developing the exploit script, running `xshop test` each time they wish to view the results.

Example 5 shows a more extensive test, written with the XSHOP API, which verifies that the bug was introduced with version `1.0.1a` and patched in `1.0.1g`.

```
from xshop import test

variables = {
  'version': ['1.0.0r', '1.0.1a', '1.0.1b', '1.0.1c',
              '1.0.1d', '1.0.1e', '1.0.1f', '1.0.1g']}

T = test.Trial(variables)
T.run()
```

```
[Output]
Running Test:  {'version':  '1.0.0r'}, Safe
Running Test:  {'version':  '1.0.1a'}, Vulnerable
Running Test:  {'version':  '1.0.1b'}, Vulnerable
Running Test:  {'version':  '1.0.1c'}, Vulnerable
Running Test:  {'version':  '1.0.1d'}, Vulnerable
Running Test:  {'version':  '1.0.1e'}, Vulnerable
Running Test:  {'version':  '1.0.1f'}, Vulnerable
Running Test:  {'version':  '1.0.1g'}, Safe
```

Example 5: Evaluating the security of the eight OpenSSL versions against the Heartbleed vulnerability by using XSHOP.

It is clear that with minimal additional effort, an exploit author has access to a much more robust testing environment, with deterministic test results. They are able to expand the scope of tests quickly, and eliminate repetitive test environment configuration. The rest of the community enjoys immediate access to the same test environment, eliminating the task of reproducing the author's results entirely.

CHAPTER VIII

EVALUATION

In this section, we attempt to evaluate XSHOP by applying it to a number of use cases and discussing its impact in each case. With these case studies, we hope to answer the following questions:

1. What kind of real exploits can XSHOP support? (§8.1)
2. How realistic is the security benchmark suite that is built with XSHOP? (§8.2)
3. How effective is the pen-testing tool that is implemented with XSHOP? (§8.3)
4. How flexible are the hooking mechanisms that XSHOP provides? (§8.4)
5. How easy is it to write an exploit and its configuration for XSHOP? (§8.5)

8.1 Testing XSHOP *with Real Exploits*

XSHOP can handle various types of exploits because it allows for very flexible test configurations. At a minimum, the author must provide some way of detecting the results of an exploit attempt. XSHOP allows this detection to take place in any system within the test environment, allowing for a wide variety of detection techniques. For example, in control-flow hijacking vulnerabilities, the check can be done from the attacking system by examining the effect of running shellcode or from the target system by monitoring for system signals such as `SIGSEGV` or `SIGILL`. For an information leak vulnerability, the test can inspect the properties of the return data (e.g. length, value, etc.) to determine whether the attack was successful. Other types of bugs such as race conditions and logic bugs can be identified by return values as well.

Some exploits require a specific kernel version or the lack of kernel-based mitigation techniques, such as address space layout randomization (ASLR). XSHOP supports the testing of these exploits when the test environment is constructed using a virtual machine based provider. Other providers, such as Docker, offer performance advantages but can not be used for these tests.

In the rest of this section, we will describe the implementation of five different types of exploits: leaking uninitialized memory, control-flow hijacking by stack overflow, logic bugs, thread race conditions, and a kernel vulnerability. To present practicality of the work, we use real world vulnerabilities (e.g. porting Metasploit module into XSHOP) if possible.

8.1.1 Leaking Uninitialized Memory

`CVE-2013-6629` refers to a vulnerability in `libjpeg` in which uninitialized memory is leaked when the library attempts to open a specially crafted `JPEG` file. To create a proof of concept for this vulnerability, we wrote a test script based on the public disclosure [49] which attempts to open and read such a crafted file. The library is considered to be vulnerable if it returns image data as 16KB without error. This returned data is partially comprised of uninitialized areas of memory. XSHOP was able to verify that versions 6b and 8 are vulnerable, while version 9 correctly patched the vulnerability.

8.1.2 Control-flow Hijacking

Control-flow hijacking vulnerabilities are very common, and the most critical attack vector in the wild [12]. To demonstrate XSHOP's ability to test such attacks, we selected `CVE-2010-4221`, a vulnerability in `ProFTPD`.

`CVE-2010-4221` refers to a stack overflow in `ProFTPD` which is triggered by a large number of Telnet IAC commands. In the default configuration, SSP (Stack Smashing Protector) needs to be bypassed. Since a stack cookie has 24bits of entropy in a 32bit machine, and a stack cookie never changes until restarting the program, an exploit is possible. For simplicity, we compiled `ProFTPD` without SSP and checked for a crash by

monitoring the syslog. We considered a crash with `SIGSEGV` at the controlled `eip` to indicate exploit success. An existing exploit is taken from a Metasploit module [15]. XSHOP can verify `CVE-2010-4221` affects `ProFTPD` version 1.3.2c.

In addition to the aforementioned example, we added another control-flow attack example based on overwriting a `C++` virtual function pointer. The example simulates a typical use-after-free (UAF) attack, as it calls a different function than what it is supposed to call when it uses the pointer. The example creates a `C++` object and modifies the object's virtual function table with a fixed string printing function. It then triggers the modified virtual function. If virtual function overwriting is possible, the program will output the pre-defined string. By checking this, we can determine whether the exploit succeeds or not. We wrote an exploit for the example program, and tested it using XSHOP. We will use this example for demonstrating how a compiler-based control-flow integrity (CFI) (e.g. clang CFI) mitigation techniques can be tested on XSHOP, in §8.4.

8.1.3 Logic Bugs

To demonstrate XSHOP's ability to verify logic bug style attacks, we selected the Shellshock family of bugs. This group of vulnerabilities allows an attacker to execute arbitrary commands on an unpatched machine by taking advantage of bugs in how GNU Bash interprets commands. We note that Shellshock was an extremely critical vulnerability when first discovered and makes for a great example of XSHOP's utility. Bash possesses a wide range of versions and patches, many of which are used by various distributions.

We show how XSHOP can be used to produce a comprehensive set of tables which indicate the vulnerability status of every combination of version, patch and CVE, automatically and with minimal setup. In total, XSHOP built 262 different patch versions in a fully automated fashion. Our results are presented in Figure 3.

The analysis shows that the CVEs for Shellshock were not all patched simultaneously, and patches were applied in the same manner across all versions (see the pattern of edges in red color). In addition to demonstrating support for testing implementation bugs, this

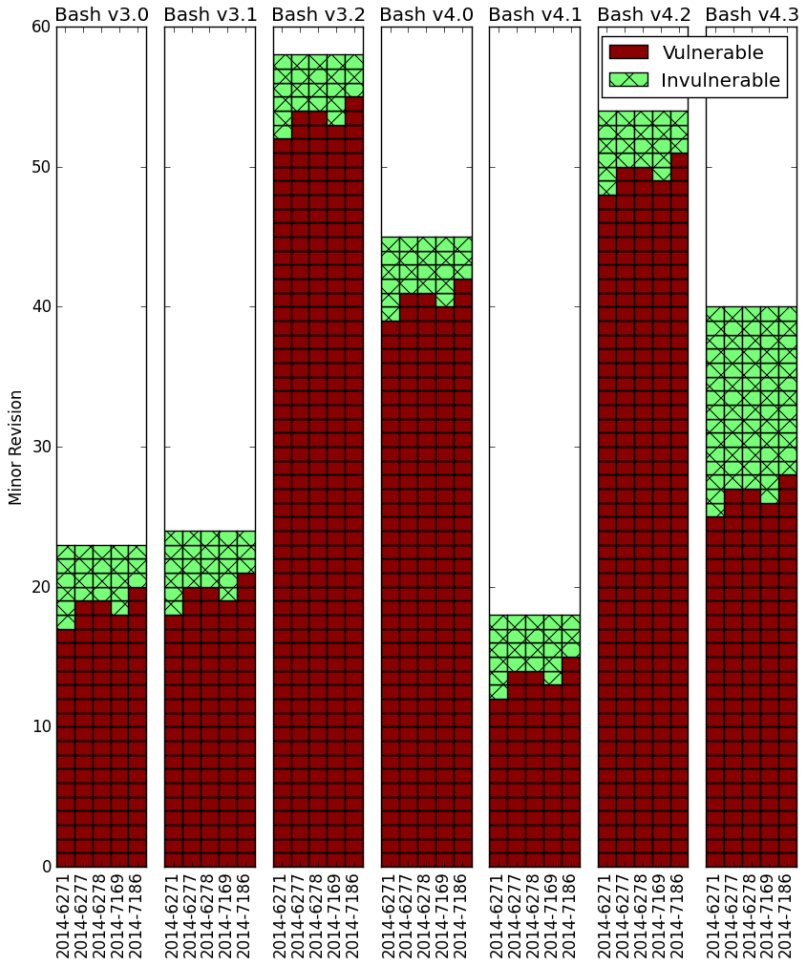

Figure 3: Shellshock vulnerability by Bash version (horizontal axis) and patch (minor version in vertical axis) for five major associated CVEs (sub-horizontal axis). The patterns of vulnerable versions (colored in red) show that the vulnerability is successfully patched in such sequence, in the same manner, regardless of major version.

evaluation gives some idea of the scale of testing that can be facilitated by XSHOP.

8.1.4 Race Conditions

A less common type of vulnerability involves the attacker taking advantage of a race condition between threads in a multi-threaded program. As an example, we wrote a simple program in which two threads are created which both attempt to write to a global variable. We were able to verify that both threads executed in the test environment and were able to modify the global variable, leading to unpredictable behavior. This race condition example is explored further in §8.2.

8.1.5 Kernel Vulnerabilities

To confirm XSHOP's ability to evaluate kernel exploits, we wrote PoC tests for the "Full Nelson" and "Half Nelson" privilege escalation attacks. To determine the success of the exploit, we use the results of `getuid()`, in the same manner as the original exploit, but modify the script to return `XSHOP_SUCCESS` or `XSHOP_FAILURE` instead of launching a shell. We switch to Vagrant for the isolation provider, and begin with an unpatched Ubuntu 10.10 Server box. To perform each test, the box is upgraded or downgraded to the required kernel version during the build phase when launching the test environment. XSHOP is able to verify the posted version number in which the vulnerability was patched in the kernel.

8.2 Evaluating Mitigation Schemes

To illustrate XSHOP's capabilities at managing multiple independent variables during testing, we chose to evaluate various mitigation techniques offered by the `Clang` compiler. These techniques are applied to some examples from the previous section to determine their effectiveness.

To expand these vulnerability tests, a new dimension is added: compiler flag. We place a hook which passes the flag for a given test to the compiler during the build phase. This hook either instructs `Clang` to compile normally, or with the mitigation technique to be tested.

8.2.1 AddressSanitizer and SafeStack

To test the defense capability of AddressSanitizer [44] and SafeStack [25] against `CVE-2010-4221`, we setup XSHOP to build a vulnerable version of `ProFTPD` with appropriate compiler flags, `-fsanitize=address` and `-fsanitize=safe-stack`, to apply the protection. Since the vulnerability is a typical stack buffer overflow, both mitigations successfully defend against the vulnerability. AddressSanitizer crashes the program after detecting the stack overflow vulnerability. For SafeStack, as it uses the correct return address even if the value is overwritten, the program worked as if no attack took place.

8.2.2 Memory Sanitizer

To test the ability of MemorySanitizer [47] to prevent reads of out of bounds memory, we invoked it when compiling `libjpeg` for testing `CVE-2013-6629`. During the test, the library begins to read the image as normal, and MemorySanitizer crashes the program when it attempts to read the first out of bounds byte of memory. This shows that MemorySanitizer is very effective at explicitly preventing invalid reads. As a result, compiling with `-fsanitize=memory` prevents the attack described in `CVE-2013-6629`.

8.2.3 Thread Sanitizer

To test if ThreadSanitizer (TSan) counters the race condition vulnerability, we used a simple example [48], a typical race condition bug of accessing a global variable on two different threads, without any lock. In the test, XSHOP automatically compiled a target program with and without the compiler flag `-fsanitize=thread`. Since TSan terminates the program with a non-zero exit code when it detects race condition, XSHOP can determine whether TSan detects the race condition. In the test, XSHOP detects that the TSan-enabled binary is terminated with a non-zero exit code, but the TSan- disabled binary was not. This shows that TSan can detect a race condition in this sample program.

8.2.4 Evaluation

As a result of this suite of tests, we are able to validate the utility of several mitigation techniques. In several cases, we can modify the XSHOP PoC for a real world exploit with only a few lines of code, and immediately see the impact that the mitigation technique has. While textbook exploit examples are certainly useful, there is also need to evaluate these techniques with real world examples, which may take advantage of edge cases or unorthodox strategies for carrying out the attack. XSHOP opens up these real world exploits to many additional research applications, with no additional effort from the original exploit author.

8.3 Adapting PoC for Penetration Testing

In this section, we evaluate XSHOP's functionality for testing real world systems or system images against existing PoC. We wish to demonstrate that PoC can be immediately used for personal penetration testing, with no additional modification.

8.3.1 Remote Vulnerability Testing

To demonstrate remote testing ability, we configured a vulnerable server on Amazon Web Services EC2 running with OpenSSL `1.0.1f` compiled from source. We then ran XSHOP to attack both this vulnerable server and an up-to-date web server. XSHOP confirmed that the Amazon instance leaked information while the patched web server was invulnerable.

8.3.2 Local Vulnerability Testing

To demonstrate testing vulnerabilities on existing images, we use XSHOP to test two systems for vulnerability to Shellshock by containerizing their existing file system. The first system is a multiuser server that is used for research which has many different software packages installed, runs Ubuntu 14.04 kernel 3.13, and is fully updated. Using XSHOP we could verify that the server was not vulnerable to any of the published CVEs related to Shellshock. The second system is a virtual machine running an unpatched Ubuntu 12.04 Live CD which was produced roughly one month prior to the Shellshock discovery. XSHOP confirmed that the system was vulnerable to all vulnerabilities listed in the Figure 3. These tests can be performed on images of production servers, production container images, or containerized clones of production file systems. They can be run very quickly, and required no modification to the existing PoC.

8.4 Evaluating CFI Schemes with XSHOP

In this section, we demonstrate the flexibility of XSHOP's hooking mechanism by using it for evaluating various control-flow-integrity (CFI) schemes: BinCFI, `Clang` CFI and `GCC` `vtable` verification.

8.4.1 BinCFI

We share the result of our empirical analysis on binary instrumentation based CFI mitigation, BinCFI, using XSHOP, to demonstrate XSHOP's support for flexible test design. To test the system, we configure XSHOP to build a sample binary with a stack overflow vulnerability, then apply BinCFI to the executable.

The objective of this analysis is to figure out the properties of possible return targets under the protection of BinCFI. We found that our sample exploit is still effective against the executable that is instrumented by BinCFI. In the sample program, we successfully change the return address of a `main()` function to an interim location of another function `evil()`, to directly call `system()` to execute the shell command. Since BinCFI is a mitigation that affects the control-flow of the program, an arbitrary change of return address is prevented according to the original developer's paper. To evaluate the property of possible return targets, we chose to exhaustively search on the return address to figure out what kind of return target is allowed from the vulnerability in the `main()` function of our sample program. We setup XSHOP to test against a BinCFI instrumented binary for each instruction address space. Since XSHOP can determine the success or failure of returning to the specified address by watching for error messages printed by BinCFI instrumentations, XSHOP can catch whether the address is allowed for a return target (runs correctly or throws segfaults on an address other than the return target) or not (failed, if segfault happens at the return target address).

From XSHOP test results, we find 14 reachable instructions, and can determine the common characteristics of this address, that those instructions immediately follow `call` instructions. As a result, we prove that, as the BinCFI paper mentioned, jumping to an inter-instruction address is not possible (as it does not show on the target list), and it allows all instructions right after a call instruction as a return target. Thus, the provided CFI property is call-return target matching.

8.4.2 Compiler-based Schemes

Compilers such as Clang and GCC provide CFI-like protection mechanisms especially to prevent virtual function table overwrite. We evaluated these mechanisms with XSHOP. Unlike the BinCFI example, we must use different binaries for each test, compiled with different compilers and flags. As a result, the `evil()` function's address is not consistent. Hooks therefore select the compiler/flag for creating the target binary (`clang++`, `g++`, `clang++` with `-fsanitize=cfi`, `g++` with `-fvtable-verify=std`), and then the target address is found and passed to the exploit script during the test procedure. Example 6 shows our results. XSHOP verifies that GCC's virtual table verification and Clang's CFI can prevent the virtual function table overwrite in the example.

```
[Output]
Running Test:  {'compiler':  'g++'}, Vulnerable
Running Test:  {'compiler':  'clang++'}, Vulnerable
Running Test:  {'compiler':  'g++ -fvtable-verify=std'}, Safe
Running Test:  {'compiler':  'clang++ -fsanitize-cfi'}, Safe
```

Example 6: XSHOP result for compiler based CFI schemes. Since the address of `evil()` varies, it needs to be extracted using a hook. The result shows that the virtual table verification in GCC and the CFI in Clang can mitigate the exploit in the example.

8.5 Simplicity

Table 4 shows the number of lines of XSHOP test code required to implement real world exploits. Since XSHOP can use any existing exploit framework, such as `Metasploit` and `CANVAS`, or compiled script, or package manager, it is not necessary to write many lines of code. Users simply have to modify the configuration (`config.yaml`), environment setup (`XShopFile`) and test (`xshop_test.py`) files to match their needs. Since `xshop_test.py` can call external modules, there is no restriction on exploit code language. For example, we used a Python script in `Heartbleed`, a compiled C program for the `GHOST` and `libjpeg` vulnerabilities, a `Metasploit` module for the `ProFTPD` vulnerability and a Bash script for `Shellshock`.

Table 4: Five real exploits that we have tested with XSHOP. The right columns, marked C, D and X, indicate additional lines of code, starting from a default project, that were required test each exploit.

T	CVE	Nickname	C.	XF.	X.
U	`CVE-2015-0235` [1]	GHOST	10	6	11
U	`CVE-2014-6271` [46]	Shellshock	20	8	9
U	`CVE-2013-6629` [49]	`libjpeg` memory leak	8	13	14
U	`CVE-2010-4221` [41]	`ProFTPD` stack overflow	10	4	13
U	`CVE-2014-0160` [39]	Heartbleed	10	2	11
K	`CVE-2010-4258` [11]	Full Nelson	8	3	8
K	`CVE-2010-4073` [21]	Half Nelson	8	3	8
K	`CVE-2010-3301` [6]	`ia32syscall` Emulation	8	3	8
K	`CVE-2010-3904` [10]	Linux `RDS` Protocol	8	3	8
K	`CVE-2012-0056` [55]	Mempodipper	8	3	8
K	`CVE-2013-2094` [2]	`perf_event_open`	5	3	10
K	`CVE-2015-1328` [42]	Ubuntu `OverlayFS`	5	3	11

C: `config.yaml`, **XF**: `XShopFile`, **X**: `xshop_test.py`,
T: Type, **K**: Kernel, **U**: Userspace

A significant portion of the time spent implementing a PoC is spent on the peculiarities of configuring the target library and determining the most reliable method for determining whether the exploit succeeded. These tasks are an unavoidable part of general exploit authoring. With XSHOP, the repetitive, manual configuration process is entirely skipped.

This evaluation demonstrates the simplicity of authoring a new PoC, and getting up to speed with a newly published PoC. With the entire project described in around 50 lines of code, many of which are common between projects, simple shell commands, or Python script, XSHOP projects are extremely transparent and introduce negligible overhead for exploit authors.

CHAPTER IX

DISCUSSIONS AND LIMITATIONS

In this section we discuss some observations from the evaluation process. In particular, we note some key things that PoC authors must keep in mind when developing with XSHOP. Finally we address any limitations which were noticed during the evaluation phase.

9.1 Observations

9.1.1 Deterministic Build Process

XSHOP is very useful for creating reliable PoCs, by eliminating a large amount of variation in the test environment. The highly consistent process for constructing the environment means that fragile exploits can still work when tested by other users.

For most cases, sourcing dependencies from a distribution's package manager is sufficient. Some exploits are especially sensitive to changes (e.g. Return-oriented Programming, ROP). Installed packages may be updated in the repository, and the author may wish to source them from a static file instead, to prevent changes in the future. The base image may be updated by the maintainer and the exploit author must supply their own. Non-determinism can be introduced at compile time, and deterministic build tactics such as those developed by the Tor Foundation [38] and the Debian project [29] must be used. All of these methods are left to the author's discretion and are fully supported by XSHOP.

9.1.2 Test Criterion

Another important consideration is that XSHOP only returns the results of the test specified by the user. If the test if flawed, then the results will be flawed. For instance, say a user wishes to evaluate a mitigation scheme against a static ROP exploit (e.g. it attempts to jump to a hard coded address). A mitigation technique which simply changes the memory layout

of the target by a few bytes will break the attack. The author must be able to recognize that the mitigation technique did not actually patch the vulnerability.

9.1.3 Incentive to Exploit Writers

The last two sections outlined a number of responsibilities which exploit writers will have when authoring PoC. For authors, XShop reduces the work involved in setting up specific test environments, but does not reduce their responsibility to ensure that their test properly validates the exploit. We would, however, like to provide some additional justification for XShop's use by exploit authors:

1. XShop does not add significant overhead to the existing exploit authoring process. Once a reliable test environment is established, it is easy to configure XShop to match.

2. XShop completely eliminates the repetitive task of provisioning the test environment, and is therefore far more useful to the community, allowing for more widespread use of the PoC.

3. XShop projects provide a more transparent view of the test environment, making debugging easier.

4. XShop test environments are isolated and freshly built for each test, preventing contamination of the test from previous tests or the host.

5. Distributing a PoC as an XShop project can reduce the number of variables which must be accounted for by the author. They are free to make the test environment as static as they like.

9.2 Limitations

9.2.1 Support for Other Operating Systems

While XShop supports the virtualization of a variety of operating systems via VirtualBox, the automated build process relies on SSH and is tailored to a Linux environment. The

modular provider interface, however, would make introduction of a provider with support for other operating systems, such as Windows, straightforward.

9.2.2 Bootstrapping Community

We believe that the real benefit that XSHOP could provide depends on adoption by the community. There are a number of advantages that would come from a large database of ready to use XSHOP exploit PoCs. Before this is possible, infrastructure is required for sharing and collaborating on the development of these PoCs. This is one major area of planned further development.

CHAPTER X

RELATED WORK

XSHOP is related to several previous works in the field, including deterministic packaging and building portable programs, system isolation mechanisms, benchmarking tools for exploit mitigation techniques, and pen-testing tools. In this section we discuss each in detail.

10.1 Deterministic Packaging and Compilation

When building victim environments, XSHOP needs a mechanism for resolving library dependencies. There are a number of preexisting works which tackle this problem. Package managers, such as `nix`, `apt`, `yum`, etc., are very mature technologies for managing the installed libraries on various distributions of Linux. As XSHOP uses a Debian container in its default test image, we applied `apt` as our default package manager. For resolving runtime dependencies, CDE [17] solved the challenge of identifying dependent libraries by system call introspection and it has the capability of automatically constructing the runtime environment. These mechanisms are orthogonal to XSHOP, that is, CDE can be used as a dependency resolver in XSHOP without any conflicts.

10.2 Benchmarking Exploit Mitigation

XSHOP can be compared to benchmarking toolkits for exploit mitigation techniques, such as RIPE [51]. RIPE provides several exploit test cases to measure the effectiveness of constructed defense mechanisms by applying the mechanism to those cases and attempting an attack. A significant difference between RIPE and XSHOP is that, while RIPE can only test the system against automatically generated samples that contain a vulnerability and its exploits, XSHOP can directly use real world exploits. From the modules of pen-testing tools (e.g. Metasploit [36], and CANVAS [19]), XSHOP can set up a test environment for

executing a full-fledged target application and test mitigation mechanisms against its real exploits.

Similar to RIPE, XSHOP is designed to support both compiler and post-compilation (e.g. binary, link time, etc.) instrumentation techniques. Examples of such supported mechanisms are, for compiler instrumentations, LLVM [26] based SoftBound+CETS [33, 34], AddressSanitizer [44], DangNull [27], SafeDispatch [20], CAVER [28], MemorySanitizer [47], ThreadSanitizer [45], CFI [50], CPI [25], etc. For the binary instrumentation tools, we present a working example with BinCFI [54], but there is no limitation to prevent support of Oxymoron [3], StackArmor [9], CCFIR [53], etc.

10.3 Pen-Testing

XSHOP can help various pen-testing tools to build environments in which to test its pre-built exploit modules. Metasploit [36] and CANVAS [19] are examples of such pen-testing suites. Traditionally, pen-testers are required to setup their environment to write or test such exploit modules, but XSHOP can setup this environment on their behalf, and then directly execute the exploit modules inside of the isolated test containers to evaluate their performance. For providing a similar pen-testing environment, there is a Linux distribution called Kali Linux [22] which offers a unified, preset environment for penetration testers. However, this only provides the environment for the attacking machine, and cannot set up example target machines.

10.4 Containers

XSHOP requires a mechanism to create an isolated environment which is independent from the configuration of the host. Virtual machines that run on top of the host can be used in this fashion. There are a number of implementations of such virtual machines, including Xen [4], KVM [24], QEMU [5], and Virtualbox [37]. These VM based environments come with a the cost of overhead, as they typically require additional time to build and launch the environment, and have multiple levels of world switch (i.e. ring -1 and 0), as well as the

inherent overhead in virtualizing the hardware that is presented to the guest OS. There are more light-weight containers based on namespace [30] isolation in Linux such as LXC [8] and Docker [13], as well as MBox [23], which provide a thinner filesystem sandbox that can configure the runtime environment of an application [52]. XSHOP implements container based isolation with Docker for most cases, but can also use VirtualBox VM with Vagrant for situations that require it.

CHAPTER XI

CONCLUSION

Proof-of-concept exploits are seldom reproducible because our community lacks a formal way to specify the tested environment or resolve dependencies. In this paper, we present XSHOP, a framework and infrastructure to describe environments or dependencies for PoC exploits in a formal way, thereby automatically resolving dependencies and constructing an isolated sandbox for exploit development and testing. On top of XSHOP, we build five practical use cases that utilize reproducible exploits of well-known bugs such as Heartbleed and Shellshock; the use cases include security benchmarking, pen-testing, exploit development, sharing PoC with the community, and regression testing, all with real exploits. We believe that the proposed practice not only brings immediate incentives to exploit writers but also has the potential to be grown as community-wide knowledge base.

REFERENCES

[1] 1N3, "Exim ESMTP 4.80 glibc gethostbyname - Denial of Service." https://www.exploit-db.com/exploits/35951/, Jan 2015.

[2] Andrea Bittau, "Linux Kernel < 3.8.9 - x86_64 perf_swevent_init Local Root Exploit." https://www.exploit-db.com/exploits/26131/, Jun 2013.

[3] Backes, M. and Nürnberger, S., "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, (San Diego, CA), Aug. 2014.

[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[5] Bellard, F., "QEMU, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference (ATC), FREENIX Track*, pp. 41–46, 2005.

[6] Ben Hawkes, "Linux Kernel < 2.6.36-rc4-git2 - x86_64 ia32syscall Emulation Privilege Escalation." https://www.exploit-db.com/exploits/15023/, Sep 2010.

[7] Bruening, D., Garnett, T., and Amarasinghe, S., "An infrastructure for adaptive dynamic optimization," in *Proceedings of the 2003 International Conference on Compiler Construction*, (San Francisco, CA), pp. 265–275, Mar. 2003.

[8] Canonical, Ltd., "Linux Containers." https://linuxcontainers.org/, 2015.

[9] Chen, X., Slowinska, A., Andriesse, D., Bos, H., and Giuffrida, C., "StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2015.

[10] Dan Rosenberg, "Linux Kernel <= 2.6.36-rc8 - RDS Protocol Local Privilege Escalation." https://www.exploit-db.com/exploits/15285/, Oct 2010.

[11] Dan Rosenberg, "Linux Kernel <= 2.6.37 - Local Privilege Escalation." https://www.exploit-db.com/exploits/15704/, Dec 2010.

[12] Dependable Systems Lab, "Code-Pointer Integrity - Dependable Systems Lab." http://dslab.epfl.ch/proj/cpi/, 2015.

[13] Docker, Inc., "Docker." https://www.docker.com/, 2015.

[14] Docker, Inc., "Dockerfile Reference." https://docs.docker.com/reference/builder/, 2015.

[15] Drake, J. J., "Proftpd 1.3.2rc3 - 1.3.3b telnet IAC buffer overflow (linux)." http://www.rapid7.com/db/modules/exploit/linux/ftp/proftp_telnet_iac, 2010.

[16] Google, "Google chrome v8 regression test." https://code.google.com/p/chromium/codesearch#chromium/src/v8/test/mjsunit/regress/, 2015.

[17] Guo, P. J. and Engler, D. R., "CDE: Using System Call Interposition to Automatically Create Portable Software Packages.," in *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, (Portland, OR), June 2011.

[18] HashiCorp, "Vagrant." https://www.vagrantup.com/, 2015.

[19] Immunity, Inc., "CANVAS Overview." http://www.immunityinc.com/products/canvas, 2015.

[20] Jang, D., Tatlock, Z., and Lerner, S., "Safedispatch: Securing c++ virtual calls from memory corruption attacks," in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2014.

[21] Jon Oberheide, "Linux Kernel < 2.6.36.2 - Econet Privilege Escalation Exploit." https://www.exploit-db.com/exploits/17787/, Sep 2011.

[22] Kali Linux, "Kali Linux | Penetration Testing and Ethical Hacking Linux Distribution." https://www.kali.org/, 2015.

[23] Kim, T. and Zeldovich, N., "Practical and effective sandboxing for non-root users," in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, (San Jose, CA), June 2013.

[24] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A., "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, (Ottawa, Canada), pp. 225–230, June 2007.

[25] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D., "Code-pointer Integrity," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, (Broomfield, Colorado), Oct. 2014.

[26] Lattner, C. and Adve, V., "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, (Palo Alto, CA), pp. 75–86, Mar. 2004.

[27] Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., and Lee, W., "Preventing use-after-free with dangling pointers nullification," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2015.

[28] Lee, B., Song, C., Kim, T., and Lee, W., "Type Casting Verification: Stopping an emerging attack vector," in *Proceedings of the 24th USENIX Security Symposium (Security)*, (Washington, DC), Aug. 2015.

[29] Levson, H., "Reproducible builds." https://wiki.debian.org/ReproducibleBuilds, 2015.

[30] Linux Programmer's Manual, "namespaces - overview of Linux namespaces." http://man7.org/linux/man-pages/man7/namespaces.7.html, 2015.

[31] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K., "Pin: building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, vol. 40, pp. 190–200, 2005.

[32] Mozilla, "Mozilla firefox tracemonkey regression test." http://mxr.mozilla.org/mozilla-central/source/js/src/jit-test/tests/auto-regress/, 2015.

[33] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S., "SoftBound: highly compatible and complete spatial memory safety for C," *ACM Sigplan Notices*, vol. 44, pp. 245–258, 2009.

[34] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S., "CETS: compiler enforced temporal safety for C," *ACM Sigplan Notices*, vol. 45, pp. 31–40, 2010.

[35] National Cyber Security Center Finland, "Vulnerability in the OpenSSL library requires immediate actions from web service providers." https://www.viestintavirasto.fi/en/cybersecurity/alerts/2014/varoitus-2014-01.html, Apr 2014.

[36] O'Gorman, J., Kearns, D., and Aharoni, M., *Metasploit: the penetration tester's guide*. No Starch Press, 2011.

[37] Oracle, Inc., "Oracle VM Virtualbox." https://www.virtualbox.org/, 2015.

[38] Perry, M., "deterministic builds." https://blog.torproject.org/category/tags/deterministic-builds, 2015.

[39] prdelka, "Heartbleed OpenSSL - Information Leak Exploit (1)." https://www.exploit-db.com/exploits/32791/, Apr 2014.

[40] Qualys, "Qualys Security Advisory CVE-2015-0235." https://www.qualys.com/2015/01/27/cve-2015-0235/GHOST-CVE-2015-0235.txt, 2015.

[41] R-73eN, "ProFTPd 1.3.5 (mod_copy) - Remote Command Execution." https://www.exploit-db.com/exploits/36803/, Apr 2015.

[42] rebel, "Ubuntu 12.04, 14.04, 14.10, 15.04 - overlayfs Local Root (Shell)." https://www.exploit-db.com/exploits/37292/, Jun 2015.

[43] Ronacher, A., "Jinja2." http://jinja.pocoo.org/, 2015.

[44] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D., "AddressSanitizer: A Fast Address Sanity Checker.," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, (Boston, MA), pp. 309–318, June 2012.

[45] Serebryany, K. and Iskhodzhanov, T., "ThreadSanitizer: Data Race Detection in Practice," in *Proceedings of the 2009 Workshop on Binary Instrumentation and Applications*, (New York, NY, USA), pp. 62–71, Dec. 2009.

[46] Shellshocker, "Shellshock BASH Vulnerability Tester." https://shellshocker.net/, Oct 2015.

[47] Stepanov, E. and Serebryany, K., "MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++," in *Proceedings of the 2015 International Symposium on Code Generation and Optimization*, (San Francisco, CA), pp. 46–55, Feb. 2015.

[48] Team, T. C., "Thread sanitizer." http://clang.llvm.org/docs/ThreadSanitizer.html#usage, 2015.

[49] The Hacker Factor Blog, "JPEG Patches." http://www.hackerfactor.com/blog/index.php?/archives/2014/01/11.html, Jan 2014.

[50] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G., "Enforcing Forward-edge Control-flow Integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, (San Diego, CA), Aug. 2014.

[51] Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., and Joosen, W., "RIPE: Runtime Intrusion Prevention Evaluator," in *Proceedings of the 27th Annual Computer Security Applications Conference*, (Orlando, Florida, USA), pp. 41–50, Dec. 2011.

[52] Xu, M., Jang, Y., Xing, X., Kim, T., and Lee, W., "UCognito: Private Browsing without Tears," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, (Denver, Colorado), Oct. 2015.

[53] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W., "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, (San Francisco, CA), pp. 559–573, May 2013.

[54] Zhang, M. and Sekar, R., "Control flow integrity for COTS binaries.," in *Proceedings of the 22th USENIX Security Symposium (Security)*, (Washington, DC), pp. 337–352, Aug. 2013.

[55] zx2c4, "Linux Kernel <= 2.6.39 (32-bit & 64-bit) - Mempodipper Local Root (1)." https://www.exploit-db.com/exploits/18411/, Jan 2012.